UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND
20742

LEVEL II

(12)

14 TR-898

15 AFOSR-77-3271

11 Apr[redacted]80

(12) 14

6 # FAST LANGUAGE ACCEPTANCE BY SHRINKING CELLULAR AUTOMATA

10 Azriel Rosenfeld
Angela Y. Wu*
Tsvi Dubitzki

Computer Vision Laboratory
Computer Science Center
University of Maryland
College Park, MD 20742

ABSTRACT

When bounded cellular automata are used as acceptors for formal languages, the number of time steps required to accept a string σ is at least |σ|, except in certain trivial cases, since the distinguished cell's state after t steps cannot depend on the initial states of the cells at distances > t from it. However, if the automaton is allowed to shrink (i.e., cells are deleted, and their predecessors become directly connected to their successors), language acceptance in less than linear time becomes possible.

* Also with the Department of Mathematics, University of Maryland Baltimore County, Catonsville, MD 21228.

The support of the U.S. Air Force Office of Scientific Research under Grant AFOSR-77-3271 is gratefully acknowledged, as is the help of Kathryn Riley in preparing this paper.

411 074

## 1. Introduction

It is well known that bounded cellular automata can be used as acceptors for formal languages [1]. Initially, the string $\sigma$ to be accepted is input to the automaton C, one symbol per cell. If this initial configuration leads to a configuration in which the distinguished cell $C_0$ (usually taken to be the one at the left end) enters an accepting state, we say that C has accepted $\sigma$. Evidently, except in trivial cases, the time required to accept $\sigma$ is at least $|\sigma|$ steps. Indeed, the state of $C_0$ at time $t < |\sigma|$ cannot depend on the initial states of the cells at distances $> t$ from $C_0$, so that if $C_0$ accepts in time $< |\sigma|$, it has not "seen" all of $\sigma$.

We shall see in this paper that if C is allowed to shrink (i.e., cells are deleted, and their predecessors become directly connected to their successors), language acceptance in time less than $O(|\sigma|)$ becomes possible. On cellular automata that are allowed to "reconfigure" themselves see [2].

Section 2 defines the process of language acceptance by shrinking cellular automata. Section 3 gives examples of languages that can be accepted in this way in less than linear time, and Section 4 discusses possible extensions of these ideas from string languages to array, tree, and graph languages.

## 2. Acceptance

### 2.1 Parallel parsing

Let G be a grammar, which we assume for simplicity to be context-free. Conventionally, to parse a given string $\sigma$, we find a match in $\sigma$ to the right-hand side of some rule $A \to \alpha$ of G; replace this instance of $\alpha$ by A; and repeat the process. The parse succeeds if we can reduce $\sigma$ to a single S (the start symbol of G) by proceeding in this way.

One could imagine parsing $\sigma$ "in parallel" by replacing many $\alpha_i$'s by $A_i$'s simultaneously, where the rules $A_i \to \alpha_i$ need not all be the same. (See [3] on a more restricted form of parallelism in which, for a given rule $A \to \alpha$, we always replace every instance of $\alpha$ by A simulatneously.) However, if the $\alpha$'s overlap, this replacement process may not be well defined; for example, if $\beta$ is a substring of $\alpha$, and $A \to \alpha$ and $B \to \beta$ are rules, where do we put the B relative to the A when we apply both rules? This problem of overlapping rule right-hand sides can be avoided if we do not permit any rewriting in places where overlap exists; but then it becomes impossible to apply the rule $S \to SS$ to the string $S^n$ ($n \geq 3$) in parallel, since there is overlap everywhere.

A better way to avoid the overlap problem is to use a two-step process of rule application. At the first step, each symbol in the string has the option of marking itself to indicate that it is the first symbol of a right-hand side that is to be rewritten; e.g., if T is the initial symbol of $\alpha_1, \alpha_2, \ldots$, where $A_1 \to \alpha_1$, $A_2 \to \alpha_2, \ldots$ are rules, T can mark itself to indicate which one of these $\alpha$'s, if any, is to be rewritten as the corresponding A. At the second step, we check to the right of T out to the length of the chosen rule, to see whether any other symbols in that interval are also marked. If none are marked, we replace the chosen $\alpha$ (beginning at T) by A; otherwise, we do nothing to that part of the string. This process insures that overlapping right-hand sides will never be rewritten simultaneously.

To illustrate this process, suppose we want to apply the rule $S \to SS$ to the string $S^n$. Each S (except the last one) has the option of marking itself to indicate that it and its successor are to be replaced by a single S. If a given S marks itself, and the succeeding S has not marked itself, we actually do this replacement; otherwise, we do not. Suppose that each S has probability $\frac{1}{2}$ of marking itself; then the probability of being able to apply the rule at a given S is $\frac{1}{4}$. Thus when we perform this process in parallel, about $\frac{1}{4}$ of the S's are rewritten at each stage. This means that the given string $S^n$ can be expected to shrink at each stage by a factor of $\frac{3}{4}$, i.e. its successive expected lengths are (approximately) $n$, $\frac{3}{4}n$, $(\frac{3}{4})^2 n, \ldots$ The expected number of stages required to shrink $S^n$ to a single S is thus about $\log_{4/3} n$. Thus in this example, our parallel parsing scheme parses in less than O(n) time. Other examples will be given in Section 3.

It is not hard to see that the set of strings $L_p(G)$ that can be parsed in parallel in this way, using the rules of a given context-free grammar G, is exactly L(G). Indeed, at any stage, when the symbols make the choice as to whether they should mark themselves, it is possible that only one symbol chooses to do so, which is what would happen in an ordinary parse (one rule applied at a time); thus an ordinary parse is a special case of a parallel parse, which proves that $L(G) \subseteq L_p(G)$. Conversely, at any stage of a parallel parse, a set of non-overlapping $\alpha$'s is rewritten; this could also happen in a sequential parse, if the sequence of rule applications happens to be such that this set of coexisting $\alpha$'s is rewritten before any other rewriting takes place. Thus a parallel parse can be simulated by a sequential parse consisting of subsequences of rules involving coexisting $\alpha$'s, which proves that $L_p(G) \subseteq L(G)$.

Similar remarks apply if G is a context-sensitive grammar; here parallel rewriting would be allowed as long as the rewritten substrings do not overlap, even if the contexts overlap. Some of the examples in Section 3 will involve context-sensitive G's.

## 2.2 Shrinking bounded cellular automata

Conventionally, a (one-dimensional) bounded cellular automaton (BCA) is a string of cells (automata) of fixed length. In [2] the concept of reconfiguration of a (graph-structured) BCA was introduced; here connections are allowed to be shifted from cell to neighboring cell. For our present purposes we do not need to define a general process of reconfiguration; we only need a mechanism that permits a cell to skip over a bounded number of neighbors and connect itself directly to a cell a bounded distance away. A BCA that is allowed to do this will be called a shrinking BCA.

This shrinking concept is just what we need to implement the parallel parsing scheme defined in Section 2.1. At a given stage of the parse, each cell c checks its right-hand neighbors, out to a bounded distance, to determine which rules' right-hand sides, if any, begin at c. Cell c then marks itself to indicate its choice of which rule to apply, say $A\rightarrow\alpha$, or its choice to apply no rule. Finally, c checks its right-hand neighbors out to distance $|\alpha|$. If any of them are marked, c does nothing; if none of them are marked, c changes the symbol stored in it to A and connects itself directly to the cell just following the end of $\alpha$; the cells between c and this cell are disconnected from the BCA.

## 3. Examples

Example 1: $L_1 = \{b^{2n} | n \geq 1\}$ the set of strings of even length.

Clearly $L_1$ can be generated by the linear grammar $G_1 = \{S \rightarrow bbS | bb\}$.* Given a string $b^{2n}$ for some $n \geq 1$, using this grammar, the only possible successful parse by a shrinking cellular automaton has the following sequence of configurations: $b^{2n}$, $b^{2(n-1)}S$, $b^{2(n-2)}S, \ldots, S$. This parse takes linear time, which is not any faster than a cellular automaton without the shrinking capacity, nor is it faster than a sequntial finite state automaton. Indeed, it is not hard to see that when a linear grammar is used, parallel parsing (even if shrinking is allowed) is not faster than sequential parsing, since at each stage, successful reduction can occur only at one place where the nonterminal symbol is located.

Consider the nonlinear grammar $G_1' = \{S \rightarrow SS | bb\}$. It is easy to see that $G_1'$ generates $L_1$. Using the same analysis as in Section 2.1, our parallel parsing scheme parses in less than linear time. $G_1'$ has the disadvantage that if a b associates with the wrong neighbor, for example, if the second and third b in bbbb associate and reduce to bSb, then the parse is blocked. To make sure that each b associates with the correct neighbor, it must know if its position in the string is odd or even. This takes at least linear time.

* For simplicity, a grammar <N,T,P,S> is represented by its productions P.

Using the nonlinear grammar $G_1'' = \{S \to SS \mid bSb \mid bb\}$, our parallel parsing scheme never blocks and it takes less than linear time.

It is trivial to generalize the above to languages of the form $\{b^{in} \mid n \geq 1\}$ for any $i > 0$.

<u>Example 2</u>: $L_2$ = {strings with equal numbers of b's and c's}

Let $G_2$ be the grammar $\{S \to SS \mid bSc \mid cSb \mid bc \mid cb\}$. It is obvious that $G_2$ generates $L_2$. When a string is parsed in parallel with this grammar, b's with neighboring c's can be reduced simultaneously to S for further reduction. The time required to parse a string $\sigma = \sigma_1 \sigma_2 \ldots \sigma_n$ is proportional to $k + \log(n-k)$ where k = max{difference of numbers of b's and c's in $\sigma_1 \sigma_2 \ldots \sigma_i$ for $1 \leq i \leq n$}. Thus parsing $b^j c^j$ takes linear time and parsing $(bc)^j$ takes $1 + \log j$ time.

Using the scheme in Section 2 to avoid the overlap problem can result in no reduction in a step (even though the probability of this is very small), since it is possible for every symbol to decide not to be rewritten. For a language such as $L_2$, the overlap problem can be avoided by giving priority to $S \to bc$ over $S \to cb$, and to $S \to bSc$ over $S \to cSb$, as follows: each symbol c marks itself if it has a b as its left neighbor, and each b marks itself if it has a c as its right neighbor. At the next step, the marked pairs of b and c's are rewritten as S's using $S \to bc$, and unmarked pairs of cb's (if any) are rewritten as S's using $S \to cb$. Thus cbc is reduced to cS. Similarly, $S \to bSc$ is given priority over $S \to cSb$. This method ensures that some reduction (if possible) is done at each step.

<u>Example 3</u>: $L_3$ = {strings with equal number of a's, b's, and c's}

It is well known that $L_3$ is a context sensitive language which is not context free. To parse $L_3$ in parallel, we can reduce each pair AB into C', BC into A' and AC into B' (a and A, b and B, c and C are considered equivalent) to indicate that a c, a, or b is needed to balance the number of a, b, c's. Hence A and A', B and B', C and C' cancel each other, while A' and C', B' and C', A' and B' reduce to B, A, C. In the other situations a permutation of the symbols is done, for example, BA' becomes A'B. The shorter the runs of A's, B's, C's are in a string, the more shrinking can be done at each step and the faster the string can be parsed. Using the same analysis as in Example 2, strings of the form $a^n b^n c^n$ take linear time, but most strings take less than linear time.

<u>Example 4a:</u>(Dyck languages) $L_4$ = {strings of balanced parentheses of $\ell$ types} for some $\ell \geq 1$

The grammar $\{S \to SS | b_i S c_i | b_i c_i,\ i=1,2,\ldots\}$ generates $L_4$, where $b_i$ and $c_i$ are the corresponding left and right parentheses. It is easy to see that the time needed to parse a string of length 2n in $L_4$ is proportional to $k+\log(n-k)$ where k = level of deepest nested pair of parentheses. For $\ell=1$, $L_4$ is $L_2$ restricted to having the number of a's $\geq$ the number of b's when following the string from left to right.

Example 4b: Bracketed context free languages

These are the languages generated by context free grammars in which each rule is of the form A $[_i w]_i$ where each production has a different i (i can be regarded as the index of the rule in the grammar). Parsing a string in this language in parallel can be done deterministically since the brackets give the order of reduction and no overlap or blocking problem can occur. In fact parsing these languages is the same as parsing the Dyck languages.

## 4. Extensions

We have shown that using shrinking cellular automata, many string languages can be parsed in less than linear time. It is easy to see that this concept of a shrinking cellular automaton can readily be extended to cycle languages [4] which can be considered as strings whose two ends are connected. However, extension to other structures is difficult, as we shall now see.

If we shrink a part $\delta$ of the interior of an array and replace it by a symbol A (or a smaller subarray), there is no way to connect A to all the cells connected to $\delta$ so that the resulting structure remains an array. Even if one is willing to relax the array requirement, the degree of the cell A may not remain bounded when such shrinkings are performed repeatedly. An alternative is to allow shrinking only at the array border. However, this loses the advantages of shrinking, since it is the same as replacing the discarded parts with special symbols to indicate that the cells are no longer considered as part of the array. The same problems also exist for tree languages where the borders consist of the leaf nodes.

The potential unbounded growth of the degree of cells does not occur in strings and cycles because each cell has degree at most 2. For any graph language with an unbounded number of degree $\geq 3$ nodes (for example, binary trees), when a subgraph $\delta$ is shrunk to a node A, A should be connected to all the nodes that $\delta$ was connected to. This number can grow exponentially. Therefore, the concept of shrinking is useful for and only for graphs whose nodes have degree at most 2 except for a bounded number of them which can have arbitrary degrees.

## References

1. A. R. Smith, Cellular automata and formal languages, Proc. 11th IEEE Symp. on Switching and Automata Theory, 1970, 216-224; Real-time language recognition by one-dimensional cellular automata, J. Computer Systems Science 6, 1972, 233-253.

2. A. Y. Wu and A. Rosenfeld, Local reconfiguration of networks of processors, University of Maryland Computer Science Technical Report 730, Feb. 1979.

3. A. Rosenfeld, Isotonic grammars, parallel grammars, and picture grammars, in Machine Intelligence 6 (B. Meltzer and D. Michie, eds.), Edinburgh University Press, New York, 1971, 281-294.

4. A. Rosenfeld, A note on cycle grammars, University of Maryland Computer Science Technical Report 300, April 1974.